NumPy Basics: Arrays and Vectorized Computation

- The NumPy (https://docs.scipy.org/doc/numpy-1.13.0/reference/index.html) package provides an N-dimensional array object type, ndarray, and many vector/matrix functions.
- An ndarray object is used to represent a vector or a matrix, and provides operations to perform linear algebra required by many data science tasks

In [ ]:
```
%matplotlib inline
from __future__ import division
from numpy.random import randn
import numpy as np
np.set_printoptions(precision=4, suppress=True)
```

NumPy Array

- Can have one or higher dimension, but the one- and two-dimensional array is the most common.
  - think in terms of nested lists where the sublists all have the same sizes and contain the same numeric type of elements
- All elements in an array are the same type, typically int or double
- Many operations and functions are performed element-wise, but there are also operations on rows or on columns

Creating NumPy Arrays

- Arrays can be constructed in several ways using Array Creation Routines (https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.array-creation.html#routines-array-creation)
  - From lists
  - Use functions, such as zeros(), empty(), with initial values (0's)
  - From random values, using randn()
  - From existing array by changing the stracture, using reshape()

In [ ]:
```
arr1 = np.array([6, 7.5, 8, 0, 1])
arr1
```

In [ ]:
```
list2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
arr2 = np.array(list2)
arr2
```

In [ ]:
```
print(arr1)
print("arr1: dimensions={0}, shape={1}, dtype={2}\n".format(arr1.ndim, arr1.shape,
arr1.dtype))
print(arr2)
print("arr2: dimensions={0}, shape={1}, dtype={2}\n".format(arr2.ndim, arr2.shape,
arr2.dtype))
```

Create Special Arrays

In [ ]:
```
# Other ways to create special arrays
np.zeros(10)
```

In [ ]:
```
np.zeros((3, 6))
```

In [ ]:
```
np.empty((2, 3, 2))
```

In [ ]:
```
data = randn(2, 3)
data
```

In [ ]:
```
a = np.arange(18)
print(a.reshape(3, 6))
np.arange(18).reshape(2, 3, 3)
```

Data Types for ndarrays

The type of elements can be initialized at construction time and changed subsequently

In [ ]:
```
# Create array with a given dtype
arr1 = np.array([1, 2, 3], dtype=np.float64)
arr2 = np.array([1, 2, 3], dtype=np.int32)
print(arr1)
print(arr1.dtype)
print(arr2)
print(arr2.dtype)
```

In [ ]:
```
# Type conversion, from int to float
arr = np.array([1, 2, 3, 4, 5])
float_arr = arr.astype(np.float64)
print(arr)
print(arr.dtype)
print(float_arr)
print(float_arr.dtype)
```

In [ ]:
```
# convert float to int, truncate fraction
arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])
print(arr)
print(arr.astype(np.int32))
```

In [ ]:
```
# Convert between number and string types
numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)
numeric_strings * 10 # causes an error because string does not support arithematic
operations
```

In [ ]:
```
numeric_strings = numeric_strings.astype(float) # convert string to float
numeric_strings * 10 # no error here
```

In [ ]:
```
# Convert to the dtype of another array
int_array = np.arange(10)
calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)
b = int_array.astype(calibers.dtype)
b.dtype
```

In [ ]:
```
empty_uint32 = np.empty(8, dtype='u4')
empty_uint32
```

Operations between arrays and scalars

By default, operations are applied element-wise, but can also be applied to selected rows, columns, or any arbitrary slice.

In [ ]:
```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
print(arr)
# Element-wise arithmatic operations
print(arr * arr)
print(arr - arr)
print(1 / arr)
print(arr ** 0.5)
```

Broadcast

- When operations are applied to arrays of different sizes, broadcasting is performed implicitly on the smaller array so that its size matches the larger array.
- The smaller and the larger array must have the same number of dimensions, and difference can be within one or more dimension

In [ ]:
```
a1 = np.arange(21).reshape(7, 3) # a1 has a shape of [7, 3]
a2 = np.array([1, 0, 1]) # a2 has a shape of [1, 3]
print(a1, "\n", a2)
print(a1 + a2) # a2 is broadcasted into the shape of (7, 3) by duplicating the fir
st row
print(a1 * a2)
```

Basic indexing and slicing

- There are many ways to select different portions of an array, maybe for further operations

In [ ]:
```
# Indexing into an array
arr = np.arange(10)
print(arr)
print(arr[5])
print(arr[5:8])
arr[5:8] = 12
print(arr)
```

In [ ]:
```
# Work with slice of an array
arr = np.arange(10)
print("arr =", arr)
arr_slice = arr[5:8]
print("slice arr[5:8] =", arr_slice)
arr_slice[1] = 12345
print("change the slice =", arr_slice)
print("arr =", arr)
arr_slice[:] = 64
print("change the slice =", arr_slice)
print("arr =", arr)
```

In [ ]:
```
# Indexing 2d array
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr2d)
print(arr2d[2])
print(arr2d[0][2])
print(arr2d[0, 2])
slice_2d = arr2d[1:3, 0:2]
print(slice_2d)
np.copyto(slice_2d, np.array([[30, 30], [20, 20]]))
print(arr2d)
```

In [ ]:
```
# indexing a 3d array
arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print("arr3d =\n", arr3d)
print("shape =", arr3d.shape)
old_values = arr3d[0].copy()
print("old value =\n", old_values)
arr3d[0] = 42
print("after change, arr3d =\n", arr3d)
print("after change, old value =\n", old_values)
```

In [ ]:
```
arr3d[0] = old_values
arr3d
```

In [ ]:
```
arr3d[1, 0, 2]
```

More examples of slicing

In [ ]:
```
arr[1:6]
```

In [ ]:
```
arr2d
arr2d[:2]
```

In [ ]:
```
arr2d[:2, 1:]
```

In [ ]:
```
arr2d[1, :2] # second row, first two columns
arr2d[2, :1]
```

In [ ]:
```
arr2d[:, :1]
```

In [ ]:
```
arr2d[:2, 1:] = 0
```

Boolean indexing

- Using a Boolean expression to create a list of Boolean values, and use this Boolean list as an index to select elements from an array
- The selected elements are those at the index where the Boolean value is TRUE

In [ ]:
```
# a list serves as an index for rows
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])

# an array
data = randn(7, 4) # 7 rows and 4 columns
print(names)
print(data)
```

In [ ]:
```
print(data.shape)
data.dtype
```

In [ ]:
```
names == 'Bob' # select rows 0 and 3
```

In [ ]:
```
# Choose rows at index locations corresponding to 'Bob'  (or True)
data[names == 'Bob']
```

In [ ]:
```
data[names == 'Bob', 2:] # select rows 0 and 3, and columns 2 and 3
```

In [ ]:
```
data[names == 'Bob', 3]
```

In [ ]:
```
names != 'Bob'
data[~(names == 'Bob')] # same as using name != 'Bob'
```

In [ ]:
```
mask = (names == 'Bob') | (names == 'Will')
mask
data[mask]
```

In [ ]:
```
data[data < 0] = 0 # set negative elements to zero
data
```

In [ ]:
```
data[names != 'Joe'] = 7 # set the selected rows to 7 in every column
data
```

Fancy indexing

- Select rows or columns in any order
- Use negative index to select rows or columnc
- Use a special ix_() function to specify arbitrary ordering of original elements

In [ ]:
```
arr = np.empty((8, 4))
for i in range(8):
    arr[i] = i
arr
```

In [ ]:
```
arr[[4, 3, 0, 6]]
```

In [ ]:
```
arr[[-3, -5, -7]]
```

In [ ]:
```
# more on reshape in Chapter 12
arr = np.arange(32).reshape((8, 4))
arr
```

In [ ]:
```
arr[[1, 5, 7, 2], [0, 3, 1, 2]]
```

In [ ]:
```
arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
```

In [ ]:
```
arr[np.ix_([1, 5, 7, 2], [0, 3, 1, 2])]
```

Transposing arrays and swapping axes

- Flip rows and columns

In [ ]:
```
arr = np.arange(15).reshape((3, 5))
print("arr =\n", arr)
print("transpose of arr =\n", arr.T)
```

In [ ]:
```
#arr = np.random.randn(6, 3)
np.dot(arr.T, arr)
```

In [ ]:
```
arr = np.arange(24).reshape((2, 3, 4))
print("arr =\n", arr)
arr.transpose((1, 0, 2)) # permute shape from (2, 3, 4) to (3, 2, 4)
```

In [ ]:
```
arr.swapaxes(1, 2) # from shape of (2, 3, 4), swap 3 with 4, to get the shape of (
2, 4, 3)
```

Universal Functions

Fast element-wise array functions for one or more array

In [ ]:
```
arr = np.arange(10)
print(np.sqrt(arr))
print(np.exp(arr))
```

In [ ]:
```
x = randn(8)
y = randn(8)
print(x)
print(y)
np.maximum(x, y) # element-wise maximum
```

The function modf() returns two arrays: fraction and whole number

In [ ]:
```
arr = randn(7) * 5
print(arr)
np.modf(arr)
```

Data processing using arrays

In this example, we create a data set containing the coorinates of 1000 equally spaced points within the area bounded by (-5,-5), (-5, 5), (5, -5) and (5, 5). Then, compute and plot the value sqrt(x**2+y**2) for each point, where higher value has lighter grey level.

In [ ]:
```
points = np.arange(-5, 5, 0.01) # 1000 equally spaced points
xs, ys = np.meshgrid(points, points)
print("xs =\n", xs)
print("ys =\n", ys)
```

In [ ]:
```
from matplotlib.pyplot import imshow, title
import matplotlib.pyplot as plt
z = np.sqrt(xs ** 2 + ys ** 2)
z
plt.imshow(z, cmap=plt.cm.gray); plt.colorbar()
plt.title("Image plot of $\sqrt{x^2 + y^2}$ for a grid of values")
plt.draw()
```

Expressing conditional logic as array operations

In this example, we use basic Python and NumPy to select elements from two vectors x and y according to a Boolean conditional logic. For each pair of corresponding elements, select the one from x if the Boolean value is TRUE, otherwise, select the element from y.

In [ ]:
```
x = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
y = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
cond = np.array([True, False, True, True, False])
print("x =", x)
print("y =", y)
list(zip(x, y, cond)) # align the elements and the Boolean values
```

In [ ]:
```
# use basic Python
result = [(x if c else y)
          for x, y, c in zip(x, y, cond)] # make the selections
result
```

In [ ]:
```
# use NumPy where()
result = np.where(cond, x, y)
result
```

A condition can also be the result of a Boolean test

In [ ]:
```
arr = randn(4, 4)
print("arr =\n", arr)
np.where(arr > 0, 2, -2)
np.where(arr > 0, 2, arr) # if possitive, set to 2, otherwise keep original values
```

The following is a list of types of functions provided by NumPy for Array type

NumPy Routines

Array creation routines Array manipulation routines Binary operations String operations Datetime Support Functions Data type routines Mathematical functions with automatic domain (numpy.emath) Floating point error handling Discrete Fourier Transform (numpy.fft) Financial functions Functional programming Indexing routines Input and output Linear algebra (numpy.linalg) Logic functions Mathematical functions Matrix library (numpy.matlib) Polynomials Random sampling (numpy.random) Set routines Sorting, searching, and counting Statistics

Mathematical and statistical methods

Mathematical functions (both object and universal) include - Trigonometric functions - Rounding, - Sums, products, differences, - Exponents and logarithms, - Floating point routines, - Arithmetic operations, - Handling complex numbers, and - Miscellaneous
Statistical functions include - Order statistics, - Averages and variances, - Correlating, - Histograms

Multiple version of the functions

Many version of the same operations can be applied

- top-level functions
- functions provided by object (i.e., array itself)

In [ ]:
```
arr = np.random.randn(5, 4) # normally-distributed data
print("arr =\n", arr)
print("mean =", arr.mean())  # object method
print("mean =", np.mean(arr)) # top-level method
print("sum =", arr.sum()) # sum of the whole array
```

In [ ]:
```
arr = np.arange(10)
print("arr =\t", arr)
print("cumsum =", arr.cumsum())
```

Apply functions to rows or columns

In NumPy, axis=0 means to loop through rows, axis=1 means to loop through columns

In [ ]:
```
arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
print("arr =\n", arr)
print("row means =\n", arr.mean(axis=1)) # mean of columns in a row
print("column sum =\n", arr.sum(0)) # sum rows in each column
print("column cumsum =\n", arr.cumsum(0))
print("row cumprod =\n", arr.cumprod(1))
```

Distance among data in multi-dimensional space

Distances between two n-dimensional vectors (or data points) $X=$ and $Y=$ can be measured by

- Euclidian Distance:

$$d(X, Y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

- Manhattan Distance:

$$d(X, Y) = \sum_{i=1}^{n}|x_i - y_i|$$

In [ ]:
```
def euclidian(v1, v2):
    return np.sqrt(np.sum((v1-v2)**2))
def manhattan(v1, v2):
    return np.sum(np.abs((v1-v2)))
v1 = np.array([2, 3, 4])
v2 = np.array([7, 4, 2])
# Euclidian distance
print("Euclidian distance =", euclidian(v1, v2))
# Manhattan distance
print("Manhaton distance =",manhattan(v1, v2))
```

Methods for boolean arrays

- Boolean expression can be used to perform element-wise tests. The result will be a Boolean array.
- Additional functions, such as sum() and any(), can then be applied to test the truth values.

In [ ]:
```
# Find elements in array that is larger than 2, how name in total?
# which rows have at least one? which column has at least one?
arr = randn(100).reshape(20, 5)
print("arr =\n", arr)
print("test of special values =\n", (np.abs(arr)>2))
print('total number of special elements =', (np.abs(arr)>2).sum()) # Number of pos
itive values
print("rows with special elements =\n", (np.abs(arr)>2).any(1) )
print("cloumns with special elements =", (np.abs(arr)>2).any(0))
```

Sorting

- NumPy provides a built-in sort() function, can sort an array by rows or by columns

In [ ]:
```
arr = randn(8)
print(arr)
arr.sort()
print(arr)
```

In [ ]:
```
arr = randn(5, 3)
print(arr)
arr.sort(1) # sort each row
print(arr)
```

In [ ]:
```
# A trick to find percentile
large_arr = randn(1000)
large_arr.sort()
large_arr[int(0.05 * len(large_arr))] # 5% quantile
```

Unique and other set logic

In [ ]:
```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
np.unique(names)
```

In [ ]:
```
ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
np.unique(ints)
```

In [ ]:
```
# Use set to obtain unique values
sorted(set(names))
```

In [ ]:
```
# Use in1d() to test for membership in an 1-dimensional array
values = np.array([6, 0, 0, 3, 2, 5, 6])
np.in1d(values, [2, 3, 6])
```

File input and output with arrays

- NumPy arrays can be created by reading data from a file, and can also be stored into a file.
- The file can be either binary or text

Storing arrays on disk in binary format

In [ ]:
```
arr = np.arange(10)
np.save('some_array', arr)
```

In [ ]:
```
np.load('some_array.npy')
```

In [ ]:
```
np.savez('array_archive.npz', a=arr, b=arr)
```

In [ ]:
```
arch = np.load('array_archive.npz')
arch['b']
```

In [ ]:
```
!rm some_array.npy
!rm array_archive.npz
```

Saving and loading text files

In [ ]:
```
!cat array_ex.txt
```

In [ ]:
```
arr = np.loadtxt('ch04/array_ex.txt', delimiter=',')
arr
```

Linear algebra

In [ ]:
```
# dot product
x = np.array([[1., 2., 3.], [4., 5., 6.]])
y = np.array([[6., 23.], [-1, 7], [8, 9]])
print("x =\n", x, "\ny =\n", y)
print("x dotproduct y =\n", x.dot(y))  # equivalently np.dot(x, y)
```

dot product, transpose, inverse, determinant, matrix decomposition, etc

The numpy.linalg package has a standard set of matrix decompositions and things like inverse and determinant.

In [ ]:
```
from numpy.linalg import inv, qr
np.random.seed(12345)
X = randn(5, 5)
mat = X.T.dot(X)
print("X =\n", X, "\nX.T dot X =\n", mat, "\nInverse mat =\n", inv(mat))
print("mat dot inv(mat) =\n", mat.dot(inv(mat)))
q, r = qr(mat)
print("QR decomposition of mat, Q =\n", q, "\nR =\n", r)
```

Solve a system equations

Find solutions to system of equations

$$3x_0 + x_1 = 9$$
$$x_0 + 2x_1 = 8$$

The matrix form is $AX = B$, where

$$A = \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}, X = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}, B = \begin{bmatrix} 9 \\ 8 \end{bmatrix}$$

and the solution is given by $X = A^{-1}B$, where A^{-1} is the invers of A

In [ ]:
```
A = np.array([[3, 1],[1,2]])
B = np.array([[9], [8]])
X = inv(A).dot(B)
X
```

Random number generation

- NumPy provides random numbers taken from a given probability distribution

In [ ]:
```
# random numbers from a normal distribution
samples = np.random.normal(size=(4, 4))
samples
```

In [ ]:
```
from random import normalvariate
N = 1000000
%timeit samples = [normalvariate(0, 1) for _ in range(N)]
%timeit np.random.normal(size=N)
```

Example: Random Walks

In this example, we generate a sequence of 1000 integers, starting from 0, and the next number will be current number +1 or -1 with equal probability. We will also plot the sequence.

- First, do it using basic Python
- Then do it using NumPy array
- Finally use NumPy array to simulate 5000 random walks

In [ ]:
```
# Basic Python random walk starts at 0, +1 and -1 with equal probability
# walk 1000 steps
import random
position = 0
walk = [position]
steps = 1000
for i in range(steps):
    step = 1 if random.randint(0, 1) else -1
    position += step
    walk.append(position)

plt.plot(walk[:100])
```

In [ ]:
```
np.random.seed(12345)
```

In [ ]:
```
# Using Numby array
# at anytime, the walk is the cumulated sum at that time
nsteps = 1000
draws = np.random.randint(0, 2, size=nsteps) # random draw 0 or 1 at each step
steps = np.where(draws > 0, 1, -1)           # 0 means -1, otherwise, +1
walk = steps.cumsum()                        # find cumulative sum at each step

plt.plot(walk[:100])
```

In [ ]:
```
walk.min()
walk.max()
```

In [ ]:
```
(np.abs(walk) >= 10).argmax()
```

In [ ]:
```
# Do the same thing for 5000 random walks
nwalks = 5000
nsteps = 1000
draws = np.random.randint(0, 2, size=(nwalks, nsteps)) # draws 0 or 1
steps = np.where(draws > 0, 1, -1)                     # 0 means -1, otherwise +1
walks = steps.cumsum(1)                                # Find cumulative sum withi
n each row
walks
```

In [ ]:
```
walks.max()
```

In [ ]:
```
walks.min()
```

Finding walks that reach 30 or -30

In [ ]:
```
hits30 = (np.abs(walks) >= 30).any(1)
hits30
```

In [ ]:
```
hits30.sum() # Number that hit 30 or -30
```

In [ ]:
```
crossing_times = (np.abs(walks[hits30]) >= 30).argmax(1)
crossing_times.mean()
```

In [ ]:
```
steps = np.random.normal(loc=0, scale=0.25,
                         size=(nwalks, nsteps))
steps
```